EXACTLY SOLVING SPARSE RATIONAL LINEAR SYSTEMS VIA ROUNDOFF-ERROR-FREE CHOLESKY FACTORIZATIONS*

CHRISTOPHER LOURENCO † AND ERICK MORENO-CENTENO ‡

Abstract.
Exactly solving sparse symmetric positive definite (SPD) linear systems is a key problem in mathematics, engineering, and computer science. This paper derives two new sparse roundoff-error-free (REF) Cholesky factorization algorithms which exactly solve sparse SPD linear systems $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{Q}^{n\times n}$ and $\mathbf{x}, \mathbf{b} \in \mathbb{Q}^{n\times p}$. The key properties of these factorizations are: (1) they exclusively use integer-arithmetic and (2) in the bit-complexity model, they solve the linear system $A\mathbf{x} = \mathbf{b}$ in time proportional to the cost of the integer-arithmetic operations. Namely, the overhead related to data-structures and ancillary operations (those not strictly required to perform the factorization) is subsumed by the cost of the integer-arithmetic operations that are essential/intrinsic to the factorization. Notably, to-date our algorithms are the only exact algorithm for solving SPD linear systems with this asymptotically efficient complexity bound. Computationally, we show that the novel factorizations are faster than both sparse rational-arithmetic LDL and sparse exact LU factorization. Altogether, the derived sparse REF Cholesky factorizations present a framework to solve any rational SPD linear system exactly and efficiently.

Key words. Cholesky Factorization, Exact Factorization, Sparse Matrix Algorithms

AMS subject classifications. 15A23, 65F50, 65F05, 65G50

1. Introduction.

Solving sparse symmetric positive definite (SPD) systems of linear equations (SLEs) is a fundamental problem in mathematics, computer science, and operations research. In general, SLEs with an SPD input matrix are solved via Cholesky factorization, where the input matrix, A, is factored into the product of a lower triangular matrix, L, and its transpose; that is $A = LL^T$. For SPD matrices, (floating-point) Cholesky factorization is normwise backward stable [45, 35], meaning that the relative residual norm $||A\mathbf{x} - \mathbf{b}||/(||A||||\mathbf{x}||)$ is guaranteed to be close to machine precision. However, despite strong residual error bounds, as Golub & Van Loan point out, "small residuals do not imply high [solution] accuracy" [21]. Indeed, Cholesky factorization may fail if the input is highly ill-conditioned [45, 35, 27, 44] and Cholesky is guaranteed to fail if the singular values of the input matrix are too small [10]. That said, it is unclear how many real world matrices may cause Cholesky factorization to fail, and the correctness of commercial solvers is typically taken for granted; thus, there is currently no exact algorithm specifically tuned to provide exact solutions to SPD linear systems. Developing such algorithms is the core contribution of this paper.

1.1. Contributions.

To address the issues outlined above, this paper derives two new sparse roundoff-error-free (REF) Cholesky factorization algorithms. The two new factorizations extend the (dense) REF Cholesky factorization [16] and the sparse left-looking integer-preserving (SLIP) LU factorization [33]. Specifically, we extend the REF Cholesky factorization to the sparse case by deriving both an up-

*Submitted to the editors July 15, 2021.

Funding: This work was supported in part by the National Science Foundation under Grant No, OAC-1835499. In addition, the first author was also partially supported by the Texas A&M University Graduate Merit Fellowship and the US Naval Academy Junior NARC.

†Department of Mathematics, United States Naval Academy, Annapolis, Maryland 21402 (lourenco@usna.edu)

‡Department of Industrial and Systems Engineering, Texas A&M University, College Station, Texas 77843 (emc@tamu.edu)

This manuscript is for review purposes only.

looking and left-looking variant (two of the most common types used in practice today). In addition, we modify the key subroutine within the SLIP LU factorization to exploit symmetry, skip ancillary operations, and solve both lower triangular and lower trapezoidal linear systems in order to derive two new algorithms to compute the aforesaid factorizations in a sparse manner. These two new sparse REF Cholesky factorizations are unique in that, unlike traditional Cholesky factorization, computing the REF factor L does not require the usage of square root functions; thus they may be applied to *any* rational SPD input matrix $A \in \mathbb{Q}^{n \times n}$.

In addition, we analyze the computational complexity of both factorizations. Specifically, we show that, using a bit-complexity model, both of our algorithms solve the SPD sparse linear system $A\mathbf{x} = \mathbf{b}$ in time proportional to the cost of the integer-arithmetic operations. This means that all of the overhead related to data structures and ancillary operations is asymptotically dominated by the cost of the indispensable arithmetic operations to compute the factorization. Notably, this property is not trivial in the sparse case, as currently, to the best of our knowledge, there are only three sparse factorizations to achieve this asymptotically efficient complexity bound: Gilbert and Peierls' (floating-point) left-looking LU factorization [19], Liu's (floating-point) up-looking Cholesky factorization [31], and (exact) left-looking SLIP LU factorization [33]. It is important to note that many classic sparse matrix algorithms can not meet such asymptotic efficiency as their run times may be dominated by ancillary operations such as data manipulation, access, and/or allocations; notably this includes algorithms as fundamental (and seemingly trivial) as sparse matrix multiplication [8].

Next, we perform a computational study which shows that the derived factorizations are dramatically faster than exact unsymmetric LU factorization and rational-arithmetic LDL factorization. Finally, we benchmark the relative forward error of MATLAB sparse backslash, showing that state-of-the-art floating-point Cholesky factorization may fail for a modest subset of real-world instances; thereby demonstrating the need for our algorithms to exactly solve SPD linear systems. The code associated with the sparse REF Cholesky factorizations is freely available at https://github.com/clouren/IP_Chol.

1.2. Organization. This paper is organized as follows. Section 2 gives an overview of the background required for this paper. Section 3 derives the up-looking and left-looking REF Cholesky factorizations. Section 4 analyzes the computational complexity of both factorizations. Section 5 computationally compares the derived Cholesky factorizations with exact unsymmetric factorization and rational LDL factorization on real-world SPD instances for run time and MATLAB sparse backslash for accuracy. Finally, Section 6 concludes the paper.

2. Background. This section describes the theoretical foundation of the new factorizations and is organized as follows. Subsection 2.1 discusses Integer-preserving Gaussian elimination, the basis of our algorithms. Subsections 2.2, 2.3, and 2.4 discuss (floating point) up-looking Cholesky, left-looking Cholesky, and LDL factorization, respectively. Finally, subsections 2.5 and 2.6 review the (dense) REF Cholesky factorization and the SLIP LU factorization respectively. Note that, for the sake of simplicity, the descriptions in this section assume no row or column permutations are applied to the matrix A.

2.1. Integer-Preserving Gaussian Elimination. Integer-preserving Gaussian elimination (IPGE) is an elimination process for solving a system of linear equa-

tions $A\mathbf{x} = \mathbf{b}$. Given a full rank matrix $A \in \mathbb{Z}^{n\times n}$ and right hand side vector $\mathbf{b}$, denote the kth iteration IPGE matrix as $A^{(k)}$ for $k = 0, \ldots, n$ (where $A^{(0)} = A$). Let $a^k_{i,j}$ and ρ^k denote the individual entries of $A^{(k)}$ and the kth pivot element for $1 \le i \le n$, $1 \le j \le n$, and $0 \le k \le n$ (with $\rho^0 = 1$), respectively. Then, at iteration k, the IPGE algorithm computes the entries $a^k_{i,j}$ as follows:

$$a^k_{i,j} = \begin{cases} a^{k-1}_{i,j} & \text{if } i = k, \\ \frac{\rho^k a^{k-1}_{i,j} - a^{k-1}_{k,j} a^{k-1}_{i,k}}{\rho^{k-1}} & \text{otherwise} \end{cases} \tag{1}$$

Note that Equation (1) differs from traditional Gaussian elimination only in the denominator (in traditional Gaussian elimination, the division is by the current pivot, ρ^k, instead of the previous pivot). This seemingly minor modification leads to two key properties summarized below.

LEMMA 2.1. *The divisions in Equation* (1) *are integral [14, 4, 36].*

Lemma 2.1 has been proven in three different manners.

Both Edmonds and Bareiss prove the integrality of the IPGE entries by showing (each in a different way) that each IPGE entry is a subdeterminant of A. Specifically, by using row expansions, Edmonds [14] shows that each IPGE entry, $a^k_{i,j}$, is equivalent to a corresponding subdeterminant of A. Bareiss [4] proves the integrality of the IPGE entries using Sylvester's identity. Specifically, he shows that a block triangular partitioning of A can be manipulated to to take the form of Sylvester's identity [37], and then, uses various sub determinant properties to derive the IPGE equation. Montante-Pardo and Mendez-Cavazos [36] aim to prove the integrality of IPGE by induction; while this proof is good for building intuition, it falls short as they do not show the induction hypothesis to be true for a general k.

LEMMA 2.2. *Let* $\sigma = \max_{i,j} |a^0_{i,j}|$. *Given an SPD matrix A, the maximum bit length to store any IPGE entry, denoted* β_{max}, *is upper-bounded polynomially as follows [33]:*

$$\beta_{max} \le \lceil n \log(\sigma) \rceil.$$

Lemma 2.2 is proved as follows. Since each IPGE entry is a subdeterminant of A, its worst case bit-length can be upper-bounded polynomially. This fact was first shown in [5] and further revised in [29, 16]. The current tightest bound known for SPD matrices is the one presented in Lemma 2.2 which was derived by [33] by using Hadamard's bound for SPD matrices [23, 25]. Note that though this bound is currently the best known, it is pessimistic, as it is tight if and only if the matrix A is diagonal. That said, Lemma 2.2 is the key property making IPGE preferable to rational-arithmetic Gaussian elimination. Specifically, IPGE achieves gratis the polynomial bound on the entries; in contrast, the bit-lengths of the numerators/denominators in rational-arithmetic algorithms grow exponentially without the usage of ancillary and computationally expensive greatest common divisor operations [43, 42].

2.2. Traditional Up-Looking Cholesky Factorization. Given an SPD matrix A, the up-looking Cholesky factorization computes the sparse Cholesky factorization, $A = LL^T$, one row at a time. Graphically, Figure 1 illustrates the data access pattern of up-looking Cholesky factorization. Specifically, at iteration k, up-looking Cholesky accesses the $k-1 \times k-1$ completed portion of L (denoted $L(1 : k-1, 1 : k-1)$ and the first k entries of the kth row of A (denoted $A(k, 1 : k)$) and computes the kth row of L (thereby the kth column of L^T). Up-looking Cholesky factorization computes

the kth row of L by solving a lower triangular linear system and then computing a square root. Formally, the solutions of the equations below yield the kth row of L, where $\mathbf{x}$ is the first $k-1$ entries of this row and $l_{k,k}$ is the kth entry of the row.

$$L(1:k-1,1:k-1)\mathbf{x} = A(k:1:k-1)^T,$$

$$l_{k,k} = \sqrt{a^0_{k,k} - \mathbf{x}^T\mathbf{x}}.$$

Assuming that A has a fixed row/column ordering, the up-looking Cholesky factorization computes L in time proportional to only the number of intrinsic floating-point operations; meaning that the cost of these indispensable floating-point operations asymptotically dominates the cost of all data access, manipulation, and ancillary operations. As a result, this algorithm's computational complexity is directly proportional to its arithmetic work. Up-looking Cholesky factorization was derived in [40, 30, 3]; however, the modern version, which achieved the aforesaid asymptotic efficiency, was developed by [31].

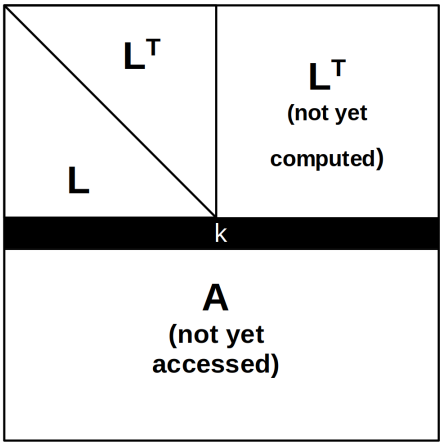


Fig. 1: Up-looking Cholesky Factorization

2.3. Traditional Left-Looking Cholesky Factorization. Given an SPD matrix A, left-looking Cholesky factorization computes the sparse Cholesky factorization, $A = LL^T$, one column at a time. Graphically, Figure 2 illustrates the data acccess pattern of left-looking Cholesky factorization. Specifically, at iteration k, left-looking Cholesky accesses rows $k,\ldots,n$ of the first $k-1$ computed columns of L (denoted $L(k:n,1:k-1)$), the kth row of L (denoted $L(k,1:k-1)$), the kth column of L (denoted $L(1:k-1,k)$) and the last $n-k$ entries of the kth column of A (denoted $A(k+1:n,k)$). Formally, the equations below give the kth column of the matrix L.

$$l_{k,k} = \sqrt{a_{k,k} - L(k,1:k-1)\cdot L(k,1:k-1)^T},$$

$$L(k+1:n,k) = (A(k+1:n,k) - L(k:n,1:k-1)L(1:k-1,k))/l_{k,k}.$$

Notice that, unlike up-looking Cholesky, left-looking Cholesky factorization is based on performing a matrix vector product (specifically, computing $L(k:n,1:$

$k-1)\cdot L(1:k-1,k)$). To the best of our knowledge, it is not claimed that this factorization meets the same asymptotically efficient complexity bound of up-looking Cholesky. Historically, left-looking Cholesky preceded the up-looking variants and can be attributed to [38, 40, 15, 18]. We conclude this subsection by noting that both left-looking and up-looking Cholesky factorization is used in the state-of-the-art package CHOLMOD [7] for solving sparse SPD systems. Specifically, up-looking Cholesky is utilized for very sparse input matrices while left-looking is used for all other matrices.

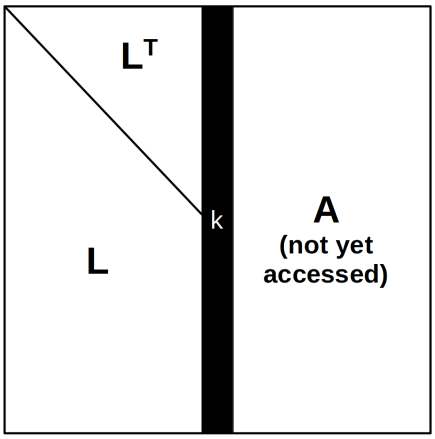


Fig. 2: Left-looking Cholesky Factorization

2.4. LDL Factorization. Given an SPD matrix A, LDL factorization computes the sparse symmetric factorization, $A = LDL^T$, where L is a unit lower triangular matrix and D is a diagonal matrix. In general, there are two variants of LDL factorization, the Bunch-Parlett LDL factorization (described below) [6] and the $L\mathbf{D}L$ factorization with a block diagonal $\mathbf{D}$ matrix. This second method, detailed in [12], is outside the scope of this paper as it is generally applied to symmetric indefinite linear systems.

The Bunch-Parlett LDL factorization exploits the symmetry of an SPD input matrix; however, in contrast to Cholesky factorization, avoids the use of square roots. Thus, LDL factorization can be used with rational-arithmetic to exactly solve SPD linear systems. Specifically, the LDL factorization generalizes Cholesky factorization since the Cholesky factor L' can be obtained from the LDL factors L and D as $L' = L\sqrt{D}$ [21].

2.5. Dense REF Cholesky Factorization. The REF Cholesky factorization solves a dense SPD system of linear equations exclusively in integer arithmetic [16]. Formally, the SPD matrix $A \in \mathbb{Z}^{n\times n}$ is factored as $A = (L\sqrt{D})(L\sqrt{D})^T = LDL^T$ where $D = diag(\rho^0\rho^1, \rho^1\rho^2, \ldots, \rho^{n-1}\rho^n)^{-1}$ and the entries in L are IPGE entries (specifically, $l_{i,j} = a_{i,j}^{\min(i,j)-1}$). Note that this factorization is indeed an integral Cholesky factorization, as the matrix D is never explicitly constructed or used when solving SPD linear systems.

2.6. SLIP LU. The SLIP LU factorization is a left-looking sparse factorization that computes the exact solution of a sparse linear system $A\mathbf{x} = \mathbf{b}$ solely using integer

arithmetic. Based on the roundoff-error-free (REF) LU factorization [16], SLIP LU computes the factorization $A = LDU$ one column at a time. At iteration k, SLIP LU accesses the first $k-1$ columns of L, denoted $L_L^{(k-1)}$, and the first $k-1$ columns of D augmented by the last $n-k+1$ columns of $(1/\rho^{k-1})I$, denoted $D_L^{(k-1)}$, and solves the following linear system:

$$L_L^{(k-1)} D_L^{(k-1)} \mathbf{x} = A(:,k).$$

The solution $\mathbf{x}$ to this linear system yields the (integral) kth column of the L and U matrices. However, this system can not be solved directly with a traditional lower triangular solve algorithm. Instead, it is solved in two phases: symbolic and numeric.

Symbolically, the nonzero pattern of the kth column of L and U is computed via a graph algorithm. Namely, this algorithm performs a sequence of depth first searches (DFS) on a graph G_L of n nodes, where a directed edge (i,j) exists if $l_{j,i} \neq 0$. The nonzero pattern, denoted $\mathcal{X}$, is given as the indices of the set of nodes reachable, via DFS, from each node j in which $a^0_{j,k} \neq 0$. This algorithm is referred to as obtaining the reach of the kth column of A on the graph of L and is denoted $\mathcal{X} = Reach_{G_L}(A(:,k))$. After $\mathcal{X}$ is computed, it is sorted prior to numeric computation.

Numerically, this system is solved via a sequence of two operations. The first, referred to as a History Update, involves multiplying the entry x_j by the current pivot and dividing it by a previous pivot which is an essential operation in order to exploit sparsity in IPGE. The second operation is referred to as an IPGE Update (Equation (1)). For completeness, this algorithm is given below as Algorithm 1.

Algorithm 1 Sparse REF Triangular Solve

```
x = A(:,k)
X = Reach_{G_L}(A(:,k))
sort(X)
Initialize history vector: h_j = 0 ∀j ∈ X
for j ∈ X do
  if j < k then
    if h_j < j − 1 then
      History update: x_j = x_j ρ^{j−1} / ρ^{h_j}
    end if
    for i > j and l_{i,j} ≠ 0 do
      if h_i < j − 1 then
        History update: x_i = x_i ρ^{j−1} / ρ^{h_i}
      end if
      IPGE update: x_i = (ρ^j x_i − l_{i,j} x_j) / ρ^{j−1}
      History vector update: h_i = j
    end for
  else
    if h_j < k − 1 then
      History update: x_j = x_j ρ^{k−1} / ρ^{h_j}
    end if
  end if
end for
```

Applied n times, Algorithm 1 factors the matrix $A \in \mathbb{Z}^{n \times n}$ by performing only

the *indispensable* number of full-precision operations. This is in contrast to rational factorization, for example, which must perform ancillary greatest common divisor operations to limit bit-growth. To date, SLIP LU is the only sparse exact factorization method for solving unsymmetric linear systems in time proportional to the arithmetic work using the bit-complexity model [33] subject to the same caveat as (floating-point) up-looking Cholesky factorization, namely that the row/column ordering is fixed.

3. Sparse REF Cholesky Factorizations. This section formally defines the two new sparse REF Cholesky factorizations, which can be applied to any rational SPD input matrix. Both factorizations are computed by repeatedly solving sparse lower triangular linear systems. Thus, in order to derive each factorization, we derive two new REF triangular solve algorithms which modify Algorithm 1 to exploit symmetry. Subsection 3.1 derives Algorithm 2, the basis of the up-looking REF Cholesky factorization, and Subsection 3.2 derives Algorithm 3, the basis of the left-looking REF Cholesky factorization. Notice that, unlike the floating-point case, the left-looking REF Cholesky factorization requires solving lower triangular systems in order to guarantee integrality when computing the factorization (due to the interrelation of consecutive pivots in IPGE). Henceforth, all matrices are assumed to be SPD, and thus, $\rho^k = a_{k,k}^{k-1}$ for all $1 \leq k \leq n$ with $\rho^0 = 1$.

3.1. Up-Looking REF Cholesky Factorization. Given an SPD matrix A, the up-looking REF Cholesky factorization computes a sparse version of the REF Cholesky factorization, $A = LDL^T$, one row at a time. At iteration k, the up-looking REF Cholesky factorization accesses the first $k-1$ rows and columns of L and D and computes the kth row of the REF Cholesky L matrix. Prior to presenting this factorization, we define the following notation:

DEFINITION 3.1. *Let $L_U^{(k)}$ and $D_U^{(k)}$ denote the kth up-looking REF L and D matrices. Specifically, $L_U^{(k)}$ and $D_U^{(k)}$ are the first k completed rows and columns of L and D, respectively.*

Theorem 3.2 gives the up-looking REF Cholesky factorization.

THEOREM 3.2. *The REF Cholesky factorization can be obtained in an up-looking fashion as follows. Initialize $\rho^1 = a_{1,1}^0$. Then, for $k = 2, \ldots, n$, solve Equation (2) and subsequently apply Equation (3).*

$$L_U^{(k-1)} D_U^{(k-1)} \mathbf{x} = A(1:k-1, k), \tag{2}$$

$$\rho^k = \rho^{k-1}(a_{k,k}^0 - \sum_{i=1}^{k-1} \frac{l_{k,i}^2}{\rho^i \rho^{i-1}}) \tag{3}$$

Prior to proving this theorem, we present and prove the following Lemma.

LEMMA 3.3. *For any $k \in \{2, \ldots, n\}$ suppose that $L_U^{(k-1)} D_U^{(k-1)}$ is correct. Then solving Equations (2) and (3) yields the kth row of L. Specifically, $\mathbf{x} = L(k, 1:k-1)^T$ and $\rho^k = L(k,k)$.*

Proof. To prove this Lemma, we will first derive an expression for entries $1:k-1$ of the kth row of L and the pivot element using the first k rows and columns of the REF Cholesky factorization, that is L_U^k and D_U^k (notice that the first $k-1$ rows and columns of this factor are $L_U^{k-1} D_U^{k-1}$). We will then show that these derived

expressions are equivalent to Equations (2) and (3), respectively. The REF Cholesky factorization of $A(1:k,1:k)$ is given as:

$$L_U^{(k)} D_U^{(k)} (L_U^{(k)})^T = A(1:k,1:k)$$

Let $\mathbf{l}_{k-1}$ and $\mathbf{a}_{k-1}$ denote entries $1:k-1$ of the kth column of $(L_U^{(k)})^T$ and A, respectively. Then, a 2×2 block decomposition of the above factorization is:

$$\begin{bmatrix} L_U^{(k-1)} D_U^{(k-1)} & 0 \\ \mathbf{l}_{k-1}^T D_U^{(k-1)} & L(k,k)D(k,k) \end{bmatrix} \begin{bmatrix} (L_U^{(k-1)})^T & \mathbf{l}_{k-1} \\ 0 & L(k,k) \end{bmatrix} = \begin{bmatrix} A(1:k-1,1:k-1) & \mathbf{a}_{k-1} \\ \mathbf{a}_{k-1}^T & a_{k,k}^0 \end{bmatrix}$$

Performing the matrix multiplication in the above decomposition, yields the following two equations:

$$L_U^{(k-1)} D_U^{(k-1)} \mathbf{l}_{k-1} = \mathbf{a}_{k-1} \tag{4}$$

$$\mathbf{l}_{k-1}^T D_U^{(k-1)} \mathbf{l}_{k-1} + L(k,k)D(k,k)L(k,k) = a_{k,k}^0 \tag{5}$$

Notice that Equation (4) is identical to Equation (2) thus $\mathbf{x} = \mathbf{l}_{k-1}$.

Likewise, recall that $D = diag(\rho^0\rho^1, \rho^1\rho^2, \ldots, \rho^{n-1}\rho^n)^{-1}$; thus $D(k,k) = \frac{1}{\rho^k \rho^{k-1}}$, $L(k,k) = \rho^k$ and

$$\mathbf{l}_{k-1}^T D_U^{(k-1)} \mathbf{l}_{k-1} = \sum_{i=1}^{k-1} \frac{L(k,i)^2}{\rho^i \rho^{i-1}}$$

Substituting the above expressions into Equation (5) and simplifying we obtain:

$$\sum_{i=1}^{k-1} \frac{l_{k,i}^2}{\rho^i \rho^{i-1}} + \frac{\rho^k}{\rho^{k-1}} = a_{k,k}^0$$

Solving for ρ^k we obtain:

$$\rho^k = \rho^{k-1}(a_{k,k}^0 - \sum_{i=1}^{k-1} \frac{l_{k,i}^2}{\rho^i \rho^{i-1}})$$

The above equation is the same as Equation (3), which completes the proof. ∎

Now we prove the correctness of Theorem 3.2.

Proof. This proof will use induction on k to show that solving Equations (2) and (3), for $k = 1, \ldots, n$, correctly obtains the REF Cholesky factorization of A in an up-looking fashion.

Base Case: $k = 1$. At step 1, we initialize $\rho^1 = l_{1,1} = a_{1,1}^0$. This is the correct first row of L.

Induction Hypothesis: For all $j < k$ assume that the solution of Equations (2) and (3) give the correct kth row of L. Then, for row k, the induction hypothesis tells us that $L^{(k-1)}D^{(k-1)}$ is correct. Lemma 3.3 showed that since $L^{(k-1)}D^{(k-1)}$ is correct, then Equations (2) and (3) correctly compute the kth row of L. Since this is true for an arbitrary k it is true for all k. ∎

Theorem 3.2 shows how to compute the up-looking REF Cholesky factorization by solving a sequence of lower triangular systems (Equation 2). However, note that, solving this system directly via a traditional (floating-point) lower triangular solve algorithm would introduce roundoff errors (this is easily noticed in Equation (3)). In order to solve these systems without roundoff errors, one could simply use SLIP LU's triangular solve algorithm (i.e., Algorithm 1). However, while avoiding roundoff errors, this approach would be inefficient for the following reasons: (1) Algorithm 1 assumes that the system is lower trapezoidal not strictly lower triangular (since it is for left-looking LU), (2) Algorithm 1 performs its symbolic analysis without exploiting the special structure of SPD instances, and (3) Algorithm 1 computes the entire row/column which would double the work for Cholesky factorization. Thus, we modify Algorithm 1 to both exploit the properties of SPD matrices and symmetry (while maintaining its integer-preserving property) as follows:

First, we modify the symbolic analysis. Since Equation (2) is a lower triangular system of the form $L^{(k-1)}D^{(k-1)}\mathbf{x} = A(1:k-1,k)$, the conditions for a nonzero entry in $\mathbf{x}$ are identical to those in the unsymmetric case, specifically: x_j is nonzero if $a_{j,k}$ is nonzero or if both $l_{i,k}$ and $l_{j,i}$ are nonzero for some i. Thus, it is valid to compute the nonzero pattern of $\mathbf{x}$ as the reach of the kth column of A on the graph of L. However, since the input matrix is SPD, the graph reachability can be computed more efficiently than in the unsymmetric case. Specifically, some of the conditions for a nonzero x_j discussed above are redundant for an SPD matrix [39]; thus, instead of analyzing the graph of L as described in Subsection 2.6, a simpler graph, referred to as the elimination tree, can be used. As its name suggests, the elimination tree of the $k-1 \times k-1$ matrix $L_U^{(k-1)}$ is an in-tree of $k-1$ nodes (i.e., a directed tree with edges directed towards the root node), where node j is the parent of node i if the first off-diagonal nonzero in column i appears in row j. Then the nonzero pattern of $\mathbf{x}$, denoted $\mathcal{X}$, is given as the set of indices of all nodes reachable while traversing (up) the elimination tree beginning at each node j in which $a_{j,k} \neq 0$ [30]. Consequently, this algorithm is referred to as obtaining the reach of the kth column of A on the elimination tree of A, and is denoted as $\mathcal{X} = Reach_{T_A}(A(1:k-1,k))$. Note that this algorithm is identical to the symbolic analysis step performed in traditional up-looking Cholesky factorization [31] due to the fact that the nonzero patterns of the REF and traditional Cholesky factorizations are identical [17].

Second, we modify the numeric routines to improve efficiency as follows. The matrices $L_U^{(k-1)}$ and $D_U^{(k-1)}$ are of dimension $k-1 \times k-1$, thus this system is indeed lower triangular and not lower trapezoidal (as is the system solved by Algorithm (1)). As a result, the up-looking triangular solve does not have to determine whether a nonzero lies above or below the current pivot element (as is done in line 6 of Algorithm 1). In addition, notice that computing Equation (3) directly is inefficient, as it requires access to previously computed entries in the kth row. Moreover, computing the pivot directly as shown is not guaranteed to preserve integrality. To remedy this issue, we note that Equation (3) is of the same format as Equation (8) of [33]; thus, as argued in that paper, it can be computed via a sequence of IPGE and History updates. To perform these operations, we update the value of ρ^k after obtaining the final value of each x_j (since each $x_j = l_{k,j}$). These numeric modifications result in performing half the number of operations as the unsymmetric variant. Taken together, the symbolic and numeric improvements lead to Algorithm 2, an efficient and exact method to compute the kth row of the integer preserving Cholesky factor L.

Algorithm 2 Symmetric Sparse REF Triangular Solve (Up-Looking)

Initialization: $x = A(1:k-1,k)$
Nonzero Pattern: $\mathcal{X} = Reach_{T_L}(A(1:k-1,k))$
sort$(\mathcal{X})$
Initialize history vector: $h_j = 0\ \forall j \in \mathcal{X}$
Initialize pivot history value: $h' = 0$
for $j \in \mathcal{X}$: $j \neq k$ **do**
  if $h_j < j-1$ **then**
    History Update: $x_j = \frac{x_j \rho^{j-1}}{\rho^{h_j}}$
  end if
  for $i > j$ **do**
    if $h_i < j-1$ **then**
      History Update: $x_i = \frac{x_i \rho^{j-1}}{\rho^{h_i}}$
    end if
    IPGE Update: $x_i = \frac{\rho^j x_i - l_{i,j} x_j}{\rho^{j-1}}$
    History Vector Update: $h_i = j$
  end for
  if $h' < j-1$ **then**
    History Update: $\rho^k = \frac{\rho^k \rho^{j-1}}{\rho^{h'}}$
  end if
  IPGE Update: $\rho^k = \frac{\rho^j \rho^k - x_j x_j}{\rho^{j-1}}$
  History Vector Update: $h' = j$
end for
if $h' < k-1$ **then**
  History Update: $\rho^k = \frac{\rho^k \rho^{k-1}}{\rho^{h_k}}$
end if

3.2. Left-Looking REF Cholesky Factorization. Given an SPD matrix A, the left-looking REF Cholesky factorization computes a sparse version of the REF Cholesky factorization, $A = LDL^T$, one column at a time. At iteration k, the left-looking REF Cholesky factorization accesses the first $k-1$ columns of L and D and computes the kth column of L. Prior to presenting this factorization, we introduce the following notation.

DEFINITION 3.4. *Let $L_L^{(k)}$ and $D_L^{(k)}$ denote the kth left-looking L and D matrices for $k = 0, \ldots, n$. Specifically, $L_L^{(k)}$ and $D_L^{(k)}$ are the first k columns of L and D augmented by I and $1/\rho^{k-1}I$, respectively.*

Theorem 3.5 formally introduces the left-looking REF Cholesky factorization.

THEOREM 3.5. *The REF Cholesky factorization can be obtained in a left-looking fashion by solving Equation 6 for $k = 1, \ldots, n$.*

$$L_L^{(k-1)} D_L^{(k-1)} \mathbf{x} = A(:,k) \tag{6}$$

Theorem 3.5 follows directly from applying Theorem 3.1 and Lemma 3.2 of [33] to an SPD matrix. However, since the left-looking REF Cholesky factorization is algorithmically different than traditional left-looking Cholesky factorization, below we present an alternate proof of Theorem 3.5. This proof illustrates that computing $\mathbf{x}$ via the traditional approach (using a matrix-vector multiplication) does not preserve

integrality because it breaks the interrelation of consecutive pivots in IPGE; thus a specially-designed REF triangular solve algorithm must be used.

Proof. We prove Theorem 3.5 via a 3×3 block matrix decomposition. The REF Cholesky factorization of an SPD matrix A is given as $A = LDL^T$. Let $L_{i,j}$, $D_{i,j}$, and $A_{i,j}$, be the i,j block of L, D, and A, respectively. At iteration k, we decompose this factorization as follows:

$$\begin{bmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{bmatrix} \begin{bmatrix} D_{1,1} & 0 & 0 \\ 0 & D_{2,2} & 0 \\ 0 & 0 & D_{3,3} \end{bmatrix} \begin{bmatrix} L_{1,1}^T & L_{2,1}^T & L_{3,1}^T \\ 0 & L_{2,2}^T & L_{3,2}^T \\ 0 & 0 & L_{3,3}^T \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{2,1}^T & A_{3,1}^T \\ A_{2,1} & A_{2,2} & A_{3,2}^T \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

where the first row and column of the above matrices represent rows/columns 1 to $k-1$, the second row and column represent row/column k, and the last row/column represent rows/columns $k+1$ to n, respectively.

Since this is a left-looking algorithm, we want to compute the kth column of L (i.e., $L_{2,2}$ and $L_{3,2}$). Via matrix multiplication, we obtain the following two equations:

$$L_{2,1}D_{1,1}L_{2,1}^T + L_{2,2}D_{2,2}L_{2,2}^T = A_{2,2}, \tag{7}$$

$$L_{3,1}D_{1,1}L_{2,1}^T + L_{3,2}D_{2,2}L_{2,2}^T = A_{3,2}. \tag{8}$$

Since $L_{2,2} = \rho^k$ and $D_{2,2} = 1/(\rho^k\rho^{k-1})$, the expression $L_{2,2}D_{2,2}L_{2,2}^T$ is equivalent to ρ^k/ρ^{k-1}. Thus, we can solve Equation (7) for ρ^k as:

$$\rho^k = \rho^{k-1}(A_{2,2} - L_{2,1}D_{1,1}L_{2,1}^T).$$

Since $A_{2,2} = a_{k,k}^0$ and $L_{2,1}D_{1,1}L_{2,1}^T = \sum_{i=1}^{k-1} \frac{l_{i,k}^2}{\rho^{i-1}\rho^i}$, the above equation becomes:

$$\rho^k = \rho^{k-1}(a_{k,k}^0 - \sum_{i=1}^{k-1} \frac{l_{i,k}^2}{\rho^{i-1}\rho^i}). \tag{9}$$

Next, since $D_{2,2}L_{2,2}^T = 1/\rho^{k-1}$, Equation (8) can be solved for $L_{3,2}$ as:

$$L_{3,2} = \rho^{k-1}(A_{3,2} - L_{3,1}D_{1,1}L_{2,1}^T). \tag{10}$$

Expanding the expression $L_{3,1}D_{1,1}L_{2,1}^T$, we obtain:

$$L_{3,1}D_{1,1}L_{2,1}^T = \begin{bmatrix} \sum_{i=1}^{k-1} \frac{l_{k+1,i}l_{k,i}}{\rho^{i-1}\rho^i} \\ \vdots \\ \sum_{i=1}^{k-1} \frac{l_{n,i}l_{k,i}}{\rho^{i-1}\rho^i} \end{bmatrix}$$

Finally, substituting the above expression into Equation (10) and solving for an arbitrary $l_{j,k} \in L_{3,2}$, we obtain:

$$l_{j,k} = \rho^{k-1}(a_{j,k}^0 - \sum_{i=1}^{j-1} \frac{l_{j,i}l_{k,i}}{\rho^{i-1}\rho^i}) \quad \text{for } j = k+1, \ldots, n. \tag{11}$$

Note that Equations (9) and (11) are of the same format as Equation (8) from [33] (with $u_{i,k} = l_{i,k}^T = l_{k,i}$ due to symmetry). Thus, as argued in that paper, in order to maintain integrality, these equations must be solved with a lower triangular solve function. ∎

Theorem 3.5 shows how to compute the left-looking REF Cholesky factorization by solving a sequence of lower trapezoidal systems (Equation 6). Similar to the discussion in Section 3.1, directly applying either a floating-point algorithm or SLIP LU's triangular solve would be inexact or inefficient, respectively. Thus, we modify Algorithm 1 to both exploit the properties of SPD matrices and symmetry (while maintaining its integer-preserving property) as follows:

Symbolically, the major difference is how the nonzero pattern $\mathcal{X}$ is computed. Computing the reach on the elimination tree (as described in Subsection 3.1) would only give the indices of the nonzeros located in the first $k-1$ rows of the kth column of L. It is also possible to compute the reach directly as is done for LU factorization; but this is overly inefficient. Instead, as is done in other modern left-looking Cholesky factorizations, we preallocate the matrix L prior to factorization [8]. Essentially, this consists of analyzing the elimination tree to compute the entire nonzero pattern of L prior to the factorization; then, the unsorted $\mathcal{X}$ is given as input to the left-looking factorization.

Numerically, we avoid half of the operations of the unsymmetric left-looking REF triangular solve. Specifically, the first $k-1$ entries of $\mathbf{x}$ are given as the kth row of L (which has already been computed). Thus the first $k-1$ entries are not recomputed and are only used to update the nonzeros located in rows $k:n$ of $\mathbf{x}$. This modification, along with the aforementioned symbolic analysis changes, lead to Algorithm 3, an efficient method to compute the kth column of the REF Cholesky factor L.

Algorithm 3 Symmetric Sparse REF Triangular Solve (Left-Looking)

Initialization: $x(1:k-1) = L(k,:)$, $x(k:n) = A(k:n,:)$
sort$(\mathcal{X})$
Initialize history vector: $h_j = 0 \; \forall j \in \mathcal{X}$
for $j \in \mathcal{X}$ **do**
  if $j < k$ **then**
    for $i > j : i \geq k$ **do**
      if $h_i < j-1$ **then**
        History Update: $x_i = \frac{x_i \rho^{j-1}}{\rho^{h_i}}$
      end if
      IPGE Update: $x_i = \frac{\rho^j x_i - l_{i,j} x_j}{\rho^{j-1}}$
      History Vector Update: $h_i = j$
    end for
  else
    if $h_j < j-1$ **then**
      History Update: $x_j = \frac{x_j \rho^{j-1}}{\rho^{h_j}}$
    end if
  end if
end for

4. Computational Complexity. This section derives the computational complexity of the up-looking and left-looking REF Cholesky factorizations. Prior to deriving these complexities, we introduce the following lemma.

LEMMA 4.1. *Based on the best known fast Fourier transform algorithm, the cost of performing multiplications and divisions on two integers of bit length β is given by $\mathcal{O}(\beta \log \beta \log\log \beta)$ [28, 41].*

Also, given an SPD matrix $A \in \mathbb{Z}^{n\times n}$ with largest initial entry, σ, Lemma 2.2 implies that the maximum bit-length required to store any entry in the REF Cholesky factorizations, denoted β_{max}, is:

$$\beta_{max} \leq n \log \sigma.$$

Prior to discussing the complexities of the sparse REF Cholesky factorizations, we note that, in contrast to the dense case, there are three special considerations common to all sparse matrix factorization complexity analyses:

1. Most sparse algorithms preorder the input matrix, A. In the pre-ordering process, A is analyzed and its columns/rows are ordered to reduce fill/operations. Critically, this problem is NP-Hard; consequently, it is solved via heuristics. Most importantly, sparse factorization complexity analysis assumes that the order of the rows and columns of A is fixed (that is, the cost of symbolic preordering is never included in the complexity analysis of a sparse factorization).
2. Sparse algorithms rely more heavily on manipulating intricate data structures; indeed, poor data manipulation or failure to exploit special objects like the elimination tree can cause the algorithm to have the same asymptotic complexity as a dense algorithm [19, 8, 13].
3. The number of arithmetic operations required to factorize a sparse matrix, A, does not depend *exclusively* on the matrix' dimension, n, but it heavily depends on the number and structure of nonzeros in A. For example, in floating-point arithmetic, if A is tridiagonal it requires $\mathcal{O}(n)$ arithmetic operations; however, if A is dense it requires $\mathcal{O}(n^3)$ arithmetic operations.

Thus, in order to better capture the amount of work, and, more precisely, in order to better distinguish between the (sparsity-specific) overhead of data structure and the necessary (arithmetic) work, the complexity of sparse matrix algorithms is typically given in terms of the number of indispensable floating-point arithmetic operations performed, denoted f [8, 13]. For the examples above $f = n$ if A is tridiagonal and $f = n^3$ if A is dense. Thus, f is a convenient term to encapsulate the number of *arithmetic* operations performed by a sparse matrix algorithm (each of which, in the floating-point case, require $\mathcal{O}(1)$ work).

Sparse matrix algorithms whose complexity depends solely on the cost of the arithmetic operations are called "proportional to arithmetic work" meaning that the cost of all ancillary operations is asymptotically dominated by the cost of the indispensable arithmetic work. For example, both Gilbert and Peierl's left LU [19] and up-looking Cholesky [31] have complexities of $\mathcal{O}(f)$ meaning that the cost of the arithmetic work asymptotically dominates the cost of all other operations. Conversely, an algorithm such as right-looking factorization performs ancillary data manipulation which is not asymptotically dominated by the cost of the arithmetic operations. Lastly, though floating-point left-looking LU and up-looking Cholesky are asymptotically efficient, their rational-arithmetic counterparts are not due to their usage of ancillary greatest common divisor operations that are required to limit the bit-length growth of entries.

To keep with this convention, we define two new terms: I_U and I_L which denote the total number of integer-arithmetic operations required to compute the sparse up-looking and left-looking REF Cholesky factorizations, respectively. Subsection 4.1 presents and proves the computational complexities of both factorizations using these terms. Then, Subsection 4.2 provides bounds on I_L and I_U to provide intuition behind the meaning of these terms.

4.1. Sparse REF Cholesky Complexities. Theorems 4.2 and 4.3 formally give the computational complexity of each factorization.

THEOREM 4.2. *The computational complexity of the sparse up-looking REF Cholesky factorization is*

$$\mathcal{O}(I_U[\beta_{max} \log \beta_{max} \log\log \beta_{max}]).$$

THEOREM 4.3. *The computational complexity of the sparse left-looking REF Cholesky factorization is*

$$\mathcal{O}(I_L[\beta_{max} \log \beta_{max} \log\log \beta_{max}]).$$

Below, we prove Theorem 4.2. Since the proof of Theorem 4.3 is very similar, it is omitted for brevity. Prior to proving Theorem 4.2, we present and prove the following Lemma.

LEMMA 4.4. *Let $\hat{I}_U$ denote the total number of integer-arithmetic operations performed by the* up-looking REF triangular solve*, Algorithm 2. The computational complexity of this algorithm is*

$$\mathcal{O}(\hat{I}_U[\beta_{max} \log \beta_{max} \log\log \beta_{max}]).$$

Proof. For Cholesky factorization, the nonzero pattern, $\mathcal{X}$, is obtained by analyzing the elimination tree; thus lines 1, 2, 3, and 4 have complexities $\mathcal{O}(|A(:,k)|)$, $\mathcal{O}(|\mathcal{X}|)$, $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$, and $\mathcal{O}(|\mathcal{X}|)$, respectively. Thus, the total cost of lines 1-4 is $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$. Note that $|\mathcal{X}| \leq n$ because the vector is reused in between iterations and thus stays a constant size n throughout the factorization.

Now, we analyze lines 5-24. As argued in Subsection 3.1 and in [19, 33], each IPGE operation corresponds to an equivalent Gaussian elimination operation and the triangular solve performs only the necessary number of operations, making these lines require $\mathcal{O}(\hat{I}_U)$ total operations. Thus, the complexity for these lines is the total number of operations multiplied by their cost, which is $\mathcal{O}(\hat{I}_U[\beta_{max} \log \beta_{max} \log\log \beta_{max}])$.

To conclude the proof, we now show that the cost $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$ is dominated by the cost of operations $\mathcal{O}(\hat{I}_U[\beta_{max} \log \beta_{max} \log\log \beta_{max}])$. Since $\beta_{max} = n \log \sigma$, we have:

$$\mathcal{O}(\hat{I}_U[\beta_{max} \log \beta_{max} \log\log \beta_{max}]) = \mathcal{O}(\hat{I}_U[n \log \sigma \log(n \log \sigma) \log\log(n \log \sigma)]).$$

Thus, since $|\mathcal{X}| \log |\mathcal{X}| \leq n \log n$, the complexity of Algorithm 2 is indeed:

$$\mathcal{O}(\hat{I}_U[\beta_{max} \log \beta_{max} \log\log \beta_{max}]).$$ ∎

We now prove Theorem 4.2.

Proof. This complexity follows from Lemma 4.4. Specifically, the overall factorization consists of repeated triangular solves; thus, the complexity of its arithmetic operations is equal to the total number of operations performed, I_U, multiplied by their cost. In addition, prior to factorization, the vectors $\mathcal{X}$, $\mathbf{h}$, and $\mathbf{x}$ are initialized requiring $\mathcal{O}(n)$. This leads to a complexity of $\mathcal{O}(n + I_U[\beta_{max} \log \beta_{max} \log\log \beta_{max}])$. (Note that the cost of creating the elimination tree of A is asymptotically dominated by using the tree in the triangular solve [8]).

The proof is completed by noting that $\mathcal{O}(n) \subset \mathcal{O}(I_U)$ since the factorization must perform at least one operation on each diagonal entry (lines 6-8 in Algorithm 2). ∎

In a bit-complexity model, these complexities mean that both the up-looking and left-looking REF Cholesky factorizations solve the SPD linear system, $A\mathbf{x} = \mathbf{b}$, in time proportional to integer-arithmetic work. To date, these are the only exact factorizations specifically tuned for SPD linear systems with this asymptotically efficient bound.

4.2. Intuition on I_U and I_L. The complexities discussed above depend on the terms I_U and I_L which are the total number of integer-arithmetic operations required for up-looking and left-looking factorization, respectively. However, much like the term f in floating-point arithmetic, these terms can be quite vague. In this subsection, we discuss some bounds on these terms in order to both relate them to the dimension of the matrix and gain some intuition.

First, note that each of these terms is lower bounded by n. Consider the case where A is diagonal. In this case, both the up-looking and left-looking algorithms must perform one multiplication and one division on each diagonal entry in rows/columns $2 : n$ (lines 6-8 of Algorithm 2 and 14-16 of Algorithm 3, respectively). Thus, for a diagonal matrix, either algorithm must perform $2(n-1)$ operations showing that I_L and I_U are both lower bounded by $\mathcal{O}(n)$.

For an upper bound, consider the case where A is dense. In this case, there is no sparsity to be exploited and operations must be performed on every entry in the matrix. As a result, $\mathcal{O}(n^3)$ operations are performed.

This dramatic difference between the lower and upper bound provides further intuition as to why the complexity of sparse matrix algorithms are not given in terms of n. That said, for most sparse input matrices, it has been shown that the number of operations required in Cholesky factorization is in the range of $[\mathcal{O}(n^{3/2}), \mathcal{O}(n^2)]$ [20, 19, 21].

5. Computational Results. This section computationally analyzes the two new factorizations and is organized as follows. Subsection 5.1 describes the specifications of the computational study. Subsection 5.2 compares the two sparse REF Cholesky algorithms. Subsection 5.3 compares our sparse REF Cholesky factorizations to a rational-arithmetic Bunch-Parlett LDL factorization [6] as well as the unsymmetric SLIP LU factorization. Lastly, Subsection 5.4 benchmarks the relative forward error of MATLAB sparse backslash on real world SPD systems.

Throughout these computational tests, we use Dolan and Moré [11] performance profiles when comparing competing algorithms/approaches. Briefly, a performance profile is a tool which takes into account both the number of instances solved as well as the cost required to solve each instance. The performance of each algorithm corresponds to a curve on a graph, where each point on the curve is what percentage of instances (y-axis) the algorithm solved within a time-multiple (x-axis) of the fastest solution time (among all algorithms) for each instance. An important property of performance profiles is that they are insensitive to the relative difficulty among different instances (i.e., they are not biased toward easy or hard instances). This is because given an instance, all solution times are relative to the fastest solver on that instance. The simplest way to interpret performance profiles is that the highest curve on the graph corresponds to the best performing algorithm. Lastly, as is common practice, all times in the performance profiles are shifted by 1 second, thereby precluding dramatic yet potentially misleading results from instances with insignificant run time differences.

5.1. Specifications.

5.1.1. Competitor Algorithms. Our first set of tests evaluate the run times of the sparse REF Cholesky factorizations. To date, direct methods could exactly solve SPD linear systems in one of two ways: rational-arithmetic Bunch-Parlett LDL factorization or exact unsymmetric LU factorization. In [33] it was shown that the SLIP LU factorization dramatically outperforms rational LU factorization. Thus, we compare our sparse REF Cholesky factorizations to rational-arithmetic LDL factorization and the SLIP LU factorization. The code for SLIP LU [32] was obtained from https://github.com/clouren/SLIP_LU and is also hosted at www.suitesparse.com. However, since we could not find a readily available, commercial-quality rational-arithmetic LDL factorization, we used the GNU Multiple Precision Arithmetic (GMP) [22] Library rational-arithmetic data type as a class template argument for the Eigen [26] solver's LDL factorization routines. Henceforth, this rational LDL factorization is referred to as Q-LDL.

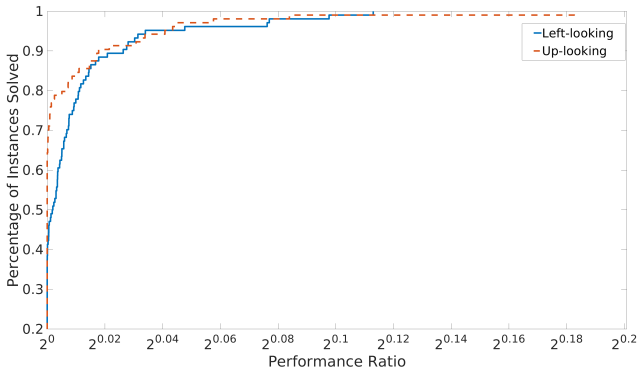
5.1.2. Computing Environment. The experiments conducted in Subsections 5.2 and 5.3 measure run time and were coded in C and performed on a computing node running CentOS 7.6.1810 which has 22 GB of RAM shared by two 2.8 GHz quad core Intel Xeon 5560 processors. The experiments conducted in Subsection 5.4 measure the accuracy of MATLAB sparse backslash and were performed in MATLAB R2020a on a computer running Ubuntu 18.04 with 16 GB of RAM using a 4.0 GHz Intel Core i7-8550U CPU. As mentioned in the previous subsection, all full precision integer and rational arithmetic operations were performed with the GMP library.

5.1.3. Chosen Instances. We tested our algorithms on a subset of the SuiteSparse Matrix Collection [9], a collection of over 2800 real-world matrices arising from a vast array of applications. To obtain the subset of SPD instances from this collection, we first selected the instances which MATLAB flagged as SPD, and then verified if they were indeed SPD using our sparse REF Cholesky factorizations. We kept a total of 104 instances: all those which were both verified to be SPD and could be factorized within 24 hours. Appendix A includes the comprehensive results along with the indices of each selected matrix from this collection.

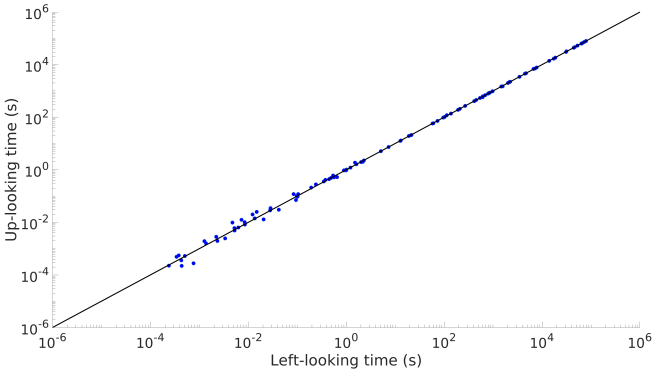
5.1.4. Symbolic Analysis. Cholesky factorization algorithms are almost always preceded by a symbolic analysis phase in which a permutation matrix P is chosen so that the Cholesky factorization of the matrix PAP^T requires less work to compute than the direct factorization of A. Finding a permutation which minimizes the number of nonzeros in the Cholesky factor is NP-Complete [46]; similarly, finding a permutation which minimizes the total work required to perform a (floating-point) Cholesky factorization is NP-Hard [34]. Exact factorization differs from these problems in that the total work to be performed is not only a function of the number of operations but also the cost of each operation (which is not constant). As a result, it is nearly certain that finding a permutation which minimizes the total work for these exact factorizations is also NP-Hard. Thus, for the symbolic analysis phase, we utilize the approximate minimum degree (AMD) ordering [1, 2], the state-of-the-art approach to efficiently reduce fill-in in the Cholesky factor. Specifically, we preordered all matrices with AMD prior to executing the exact REF or LDL factorizations.

5.2. Comparison of Sparse REF Cholesky Algorithms. This subsection compares the performance of the left-looking and up-looking REF Cholesky factorizations. Across all 104 instances the algorithms are essentially identical; the up-looking factorization's average run-time was about 0.5% smaller than the left-looking's run time, while the left-looking factorization's geometric-mean run-time was about 1%

smaller than that of up-looking. Figure 3 confirms graphically that both algorithms perform almost identically. Specifically, Figure 3a is a performance profile showing that, for all of the instances, both algorithms' run-times are within 7% of each other. Indeed, even though the profile appears to show that up-looking factorization is superior, notice that the magnitude of differences between the two algorithms (i.e., the x axis) is practically insignificant. Figure 3b further illustrates this via a scatter plot of the run-time of both algorithms for each instance. Therein, any dot lying below the line corresponds to a matrix in which left-looking required more run-time, and vice versa. Note how the only minor performance differences occurred in instances where both algorithms ran in less than 1 second.



(a) Left and Up-looking Perform Similarly



(b) Our sparse REF Cholesky factorizations are Nearly Identical

Fig. 3: Comparison of Left-looking and Up-looking Cholesky

5.3. Sparse REF Cholesky vs Alternate Direct Methods. This subsection compares the performance of the sparse REF Cholesky factorizations to both the unsymmetric SLIP LU factorization and the rational LDL factorization (Q-LDL). Since the up-looking and left-looking REF Cholesky factorizations have similar performance; for simplicity, this subsection only compares the left-looking algorithm to the alternate direct methods.

First, we compare left-looking REF Cholesky factorization to the SLIP LU fac-

torization. Since SLIP LU is an unsymmetric algorithm, we would expect it to require about double the run time of the left-looking REF Cholesky factorization. Across the 104 instances, REF Cholesky was faster than SLIP LU for 85% of the instances with an average and geometric mean run time 1.70 and 1.86 times smaller, respectively. Note that these results are slightly biased in favor of SLIP LU as it could not factor 6 of the instances within a 24 hour time period; thus its run time was set to this upper bound. Graphically, we see these results via figure 4a which gives a performance profile comparing these two algorithms which shows that left-looking Cholesky factorization clearly outperforms SLIP LU. Fix. Similarly, Figure 4b, which is a scatter plot of the algorithm's run-times, illustrates that Cholesky left-looking factorization outperforms SLIP LU on over 80% of the instances.

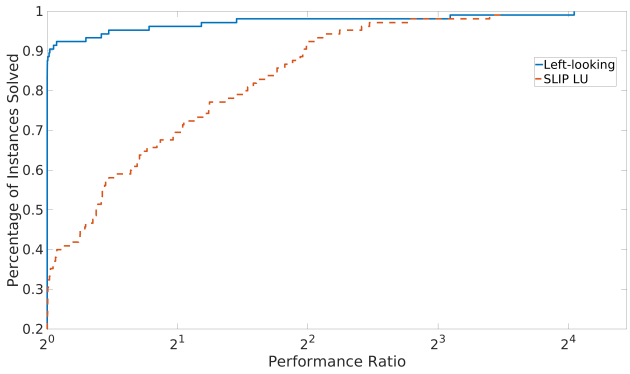
While REF Cholesky outperforms SLIP LU, its dominance is *not uniform* as expected. Specifically, SLIP LU outperforms REF Cholesky in about 15% of the test instances, a counter-intuitive result which would not occur in floating-point arithmetic. This behavior can be attributed to the interplay between the fundamental nature of SPD instances and the properties of integer-arithmetic factorization algorithms:

1. The majority of SPD instances are diagonally dominant [21] meaning that the magnitudes of their diagonal entries are larger than the sum of all other entries in their respective row/column.
2. Lourenco et al. [32] showed that IPGE-based algorithms, such as SLIP LU and REF Cholesky, perform best when small pivots are selected (specifically, in the unsymmetric case, selecting small pivots is about 4 times faster than selecting large pivots).
3. Cholesky is bound to select the (large in magnitude) diagonal entries as pivots, while SLIP LU has the freedom to select (relatively smaller) off-diagonal pivots.

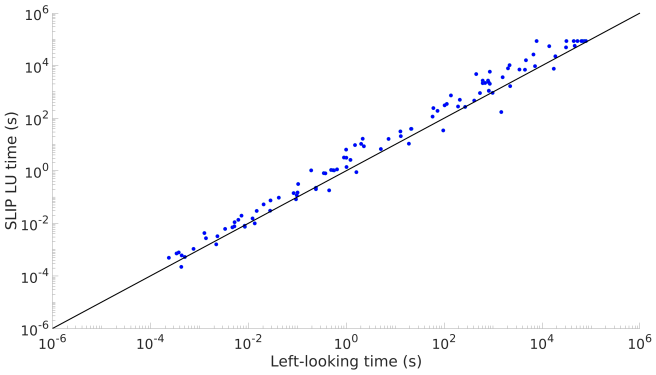
Consequently, even though SLIP LU generally performs about twice as many operations as REF Cholesky, in a small subset of instances, the smaller cost of the operations in SLIP LU allows it to negate this disadvantage. Indeed, this result provides motivation for the eventual development of an integral LDL factorization in the vein of [12] which would allow symmetric, off diagonal (thus smaller in magnitude) pivoting.

We next compare left-looking REF Cholesky factorization to rational-arithmetic LDL factorization. Across the 104 instances, REF Cholesky is faster than rational LDL on 83% of the tested instances, and its average and geometric mean run-times are 2.04 and 2.22 times smaller, respectively. Moreover, like SLIP LU, these results are slightly biased in favor of rational LDL, as this method could not factorize 8 instances within 24 hours; thus the run times for these matrices were set to this upper bound. These result support the following important observations:

1. Rational-arithmetic is incredibly slow. Notably, this can be almost exclusively attributed to the rational-arithmetic operations (and, specifically to their intrinsic GCD operations) because both REF Cholesky and rational LDL return matrices with the same nonzero pattern (both utilize AMD as a preordering and pivot along the diagonal). Conversely, the savings of REF Cholesky over rational LDL can be almost entirely attributed to its use of fast integer-arithmetic.
2. Sparse REF Cholesky factorization outperforms Q-LDL for over 80% of instances including *every* instance that requires more than two minutes except for one matrix (index 43). In contrast, the vast majority of the instances where Q-LDL is faster than left-looking REF Cholesky have insignificant run



(a) Left-looking Cholesky Outperforms SLIP LU



(b) Left-looking Cholesky Outperforms SLIP LU

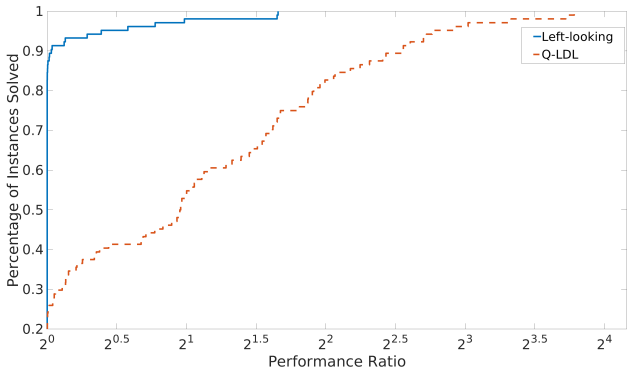
Fig. 4: Comparison of Left-looking Cholesky with SLIP LU

times for both methods (less than 1 second).

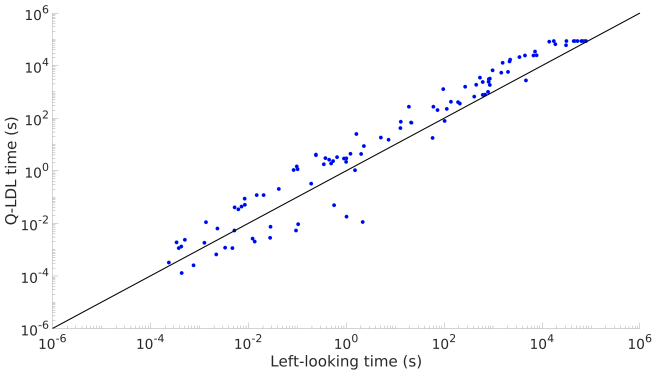
Graphically, the performance profile in Figure 5a and the scatter plot of the runtimes in Figure 5b depict the superiority of REF Cholesky over rational LDL.

We conclude this subsection with the following interesting note: Rational LDL factorization performs so poorly in general that it is slower than even the unsymmetric SLIP LU factorization. Altogether, these results present compelling evidence of the superiority of our sparse REF Cholesky factorizations over both the best available exact LU factorization (SLIP LU) and rational-arithmetic LDL factorization.

5.4. Relative Forward Error of MATLAB Sparse Backslash. As discussed in the introduction, Cholesky factorization is known to be normwise backward stable [45] meaning that the relative residual $||A\mathbf{x} - \mathbf{b}||/(||A||||\mathbf{x}||)$ is guaranteed to be in the neighborhood of machine precision. That said, a small relative residual does not necessarily mean that the solution $\mathbf{x}$ is close to the exact solution of the system [21]. In this section, we use our sparse REF Cholesky factorizations to benchmark the relative forward error of MATLAB backslash on our set of SPD matrices. For this purpose, we solve the linear system $A\mathbf{x} = \mathbf{1}$ in both MATLAB and with our sparse REF Cholesky factorization. Our exact solution is then converted to double



(a) Left-looking Cholesky Outperforms Q-LDL



(b) Left-looking Cholesky Outperforms Q-LDL in Run Time

Fig. 5: Comparison of Left-looking Cholesky with Q-LDL

precision (note that internally, this conversion is done in higher precision; thus this solution from REF Cholesky is accurate to machine precision). Finally, we compare this rounded-to-double exact solution, $\mathbf{x}^{(e)}$, to the solution returned by MATLAB, $\mathbf{x}$, by computing the relative forward error $||\mathbf{x}^{(e)} - \mathbf{x}||/||\mathbf{x}^{(e)}||$ using the 2-norm within MATLAB.

Table 1 gives the relative forward error of MATLAB sparse backslash on the tested linear systems. Overall, the Cholesky routines produce nearly exact solutions (relative forward error less than 10^{-12}) for 69% of tested instances and solutions with 6 to 12 digits, 2-5, or fewer than 2 digits of precision for 23%, 5% and 3% of instances, respectively. While these results may seem troubling, it is important to contextualize them. Specifically, it is important to note that: (1) the majority of the instances in which the MATLAB solution was not accurate have a smallest singular value in the neighborhood of machine precision (as approximately computed with MATLAB dense SVD)—thereby meeting Demmel's [10] criterion for floating-point Cholesky to fail; and (2) unlike the LU routines, the Cholesky routines within MATLAB sparse backslash do not perform any iterative refinement to fine tune the solutions.

From these results we draw two conclusions. First, enabling iterative refinement

within MATLAB Cholesky routines would likely increase the accuracy of the solutions returned. Second, exact methods are necessary for a modest subset of real world problems in which finite-precision Cholesky factorizations are not adequate.

Table 1: Relative Forward Error of MATLAB on SPD Linear Systems

Relative Forward Error Threshold	Percentage of Instances
$\leq 10^{-12}$	69.23%
$\leq 10^{-6}$	92.31%
$\leq 10^{-2}$	97.12%

6. Conclusion. This paper derives two new factorization algorithms which exactly solve sparse SPD linear systems exclusively using integer-arithmetic. Applied to a sparse matrix with a fixed row and column ordering, both derived algorithms have the property that their worst-case time complexity is proportional to the cost of the necessary integer-arithmetic work. Our derived sparse REF Cholesky factorizations require no square roots, thus they can be applied to any SPD linear system.

Computationally, we show that the derived algorithms dramatically outperform both exact LU factorization and rational LDL factorization. Specifically, REF Cholesky outperforms the unsymmetric SLIP LU factorization and the rational LDL factorization on average by a factor of 1.7 and 2.04, respectively. Surprisingly, this shows that rational LDL is even outperformed by SLIP LU. Interestingly, SLIP LU outperformed REF Cholesky on 15% of instances despite requiring twice the operations. We attribute this counter-intuitive result to the complex interplay between the number of operations performed and the cost of these operations. Specifically, unlike REF Cholesky which must select diagonal pivot elements (typically large in SPD matrices), SLIP LU is able to select pivots of small bit-length thereby reducing the cost of the exact operations. Additionally, we measured the relative forward error of MATLAB sparse backslash, showing that, as expected from the literature, it produced solutions of low accuracy on those instances which are highly ill-conditioned and/or have extremely small singular values.

Altogether, these results (1) show REF Cholesky is the best among competing factorization approaches for exactly solving SPD linear systems, (2) provide evidence that fixed precision floating-point Cholesky factorization can fail on real world highly ill-conditioned systems which do not meet the stability bounds proposed by [45, 10, 24], and (3) illustrate that commercial Cholesky routines would likely benefit from iterative refinement—a technique that is currently used in modern LU factorization routines.

In conclusion, the derived sparse REF Cholesky factorizations provide a robust framework to exactly solve SPD linear systems. The code associated with these factorizations is hosted at https://github.com/clouren/IP_Chol.

Acknowledgments. The authors would like to thank the reviewers for their very helpful comments which significantly improved the quality of this paper. The work is partially supported by the National Science Foundation under Grant No. OAC-1835499. In addition, the first author was also partially supported by the Texas A&M University's Graduate Merit Fellowship and the United States Naval Academy Junior NARC.

REFERENCES

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, 17 (1996), pp. 886–905.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff, *Algorithm 837: Amd, an approximate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software (TOMS), 30 (2004), pp. 381–388.

[3] R. E. Bank and R. K. Smith, *General sparse elimination requires no permanent integer storage*, SIAM journal on scientific and statistical computing, 8 (1987), pp. 574–584.

[4] E. H. Bareiss, *Sylvester's identity and multistep integer-preserving gaussian elimination*, Mathematics of computation, 22 (1968), pp. 565–578.

[5] E. H. Bareiss, *Computational solutions of matrix problems over an integral domain*, IMA Journal of Applied Mathematics, 10 (1972), pp. 68–104.

[6] J. R. Bunch and B. N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM Journal on Numerical Analysis, 8 (1971), pp. 639–655.

[7] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, *Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate*, ACM Transactions on Mathematical Software (TOMS), 35 (2008), p. 22.

[8] T. A. Davis, *Direct methods for sparse linear systems*, SIAM, 2006.

[9] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Transactions on Mathematical Software (TOMS), 38 (2011), p. 1.

[10] J. Demmel, *On floating point errors in Cholesky*, University of Tennessee. Computer Science Department, 1989.

[11] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical programming, 91 (2002), pp. 201–213.

[12] I. S. Duff, *Ma57—a code for the solution of sparse symmetric definite and indefinite systems*, ACM Transactions on Mathematical Software (TOMS), 30 (2004), pp. 118–144.

[13] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Oxford University Press, 2017.

[14] J. Edmonds, *Systems of distinct representatives and linear algebra*, J. Res. Nat. Bur. Standards Sect. B, 71 (1967), pp. 241–245.

[15] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman, *Algorithms and data structures for sparse symmetric gaussian elimination*, SIAM Journal on Scientific and Statistical Computing, 2 (1981), pp. 225–237.

[16] A. R. Escobedo and E. Moreno-Centeno, *Roundoff-error-free algorithms for solving linear systems via cholesky and lu factorizations*, INFORMS Journal on Computing, 27 (2015), pp. 677–689.

[17] A. R. Escobedo, E. Moreno-Centeno, and C. Lourenco, *Solution of dense linear systems vai roundoff-error-free factorization algorithms: Thoeretical connections and computational comparisons*, ACM Transactions on Mathematical Software, (2018).

[18] A. George and J. W. Liu, *Computer solution of large sparse positive definite*, Prentice Hall Professional Technical Reference, 1981.

[19] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 862–874.

[20] J. R. Gilbert and R. Schreiber, *Nested dissection with partial pivoting*, in Sparse Matrix Symposium 1982: Program and Abstracts, 1982, p. 61.

[21] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3, JHU Press, 2012.

[22] T. Granlund et al., *GNU MP 6.0 Multiple Precision Arithmetic Library*, Samurai Media Limited, 2015.

[23] J. Hadamard, *Résolution d'une question relative aux déterminants*, Bull. sci. math, 17 (1893), pp. 240–246.

[24] N. J. Higham, *Accuracy and stability of numerical algorithms*, vol. 80, Siam, 2002.

[25] R. A. Horn and C. R. Johnson, *Matrix analysis*, Cambridge university press, 2012.

[26] B. Jacob, G. Guennebaud, et al., *Eigen: C++ template library for linear algebra*, 2013.

[27] A. Kiełbasiński, *A note on rounding-error analysis of cholesky factorization*, Linear Algebra and its Applications, 88 (1987), pp. 487–494.

[28] D. E. Knuth, *The art of computer programming: sorting and searching*, vol. 3, Pearson Education, 1998.

[29] H. R. Lee and B. D. Saunders, *Fraction free gaussian elimination for sparse matrices*, Journal of symbolic computation, 19 (1995), pp. 393–402.

[30] J. W. Liu, *A compact row storage scheme for cholesky factors using elimination trees*, ACM Transactions on Mathematical Software (TOMS), 12 (1986), pp. 127–148.

[31] J. W. Liu, *A generalized envelope method for sparse factorization by rows*, ACM Transactions on Mathematical Software (TOMS), 17 (1991), pp. 112–129.
[32] C. Lourenco, J. Chen, E. Moreno-Centeno, and T. Davis, *Algorithm 1xxx: Slip lu, exactly solving sparse linear systems via a sparse left-looking integer-preserving lu factorization*. Submitted ACM Transactions on Mathematical Software, 2020.
[33] C. Lourenco, A. R. Escobedo, E. Moreno-Centeno, and T. A. Davis, *Exact solution of sparse linear systems via left-looking roundoff-error-free lu factorization in time proportional to arithmetic work*, SIAM Journal on Matrix Analysis and Applications, 40 (2019), pp. 609–638.
[34] R. Luce and E. G. Ng, *On the minimum flops problem in the sparse cholesky factorization*, SIAM Journal on Matrix Analysis and Applications, 35 (2014), pp. 1–21.
[35] J. Meinguet, *Refined error analyses of cholesky factorization*, SIAM journal on numerical analysis, 20 (1983), pp. 1243–1250.
[36] R. M. Montante-Pardo and M. A. Méndez-Cavazos, *Un método númerico para cálculo matricial*, Revista Técnico-Científica de Divulgación, 2 (1977), pp. 1–24.
[37] T. Muir and W. H. Metzler, *A Treatise on the Theory of Determinants*, Courier Corporation, 2003.
[38] Y. U. D. of Computer Science, S. Eisenstat, M. Schultz, and A. Sherman, *Efficient implementation of sparse symmetric Gaussian elimination*, 1975.
[39] S. Parter, *The use of linear graphs in gauss elimination*, SIAM review, 3 (1961), pp. 119–130.
[40] D. Rose, G. Whitten, A. Sherman, and R. E. Tarjan, *Algorithms and software for in-core factorization of sparse symmetric positive definite matrices*, Computers & Structures, 11 (1980), pp. 597–608.
[41] A. Schönhage and V. Strassen, *Schnelle multiplikation grosser zahlen*, Computing, 7 (1971), pp. 281–292.
[42] A. Schrijver, *Theory of integer and linear programming*, 1986.
[43] J. Stein, *Computational problems associated with racah algebra*, Journal of Computational Physics, 1 (1967), pp. 397–405.
[44] J.-G. Sun, *Componentwise perturbation bounds for some matrix decompositions*, BIT Numerical Mathematics, 32 (1992), pp. 702–714.
[45] J. H. Wilkinson, *A priori error analysis of algebraic processes*, in Intern. Congress Math, vol. 19, 1968, pp. 629–639.
[46] M. Yannakakis, *Computing the minimum fill-in is np-complete*, SIAM Journal on Algebraic Discrete Methods, 2 (1981), pp. 77–79.

Appendix A. Comprehensive Computational Results.

Tables 2, 3, and 4 present exhaustive computational results for all 104 tested instances sorted based on their run times. The first three columns of each table give the name (i.e., matrix index from SuiteSparse collection), dimension, and number of nonzeros of each matrix. Column 4 gives the run time of the left-looking Cholesky factorization. Finally columns 5, 6, and 7 give the run times of up-looking Cholesky, SLIP LU, and Q-LDL relative to left-looking Cholesky, respectively. Note that **N/A** in relative run times indicates that the algorithm could not factorize this instance within 24 hours. Lastly, any matrix whose solution had less than 2 digits of precision is bolded and colored **red** and any instance whose solution had between 2 and 6 digits of precision is italicized and colored *blue*.

				Relative Run Time		
Matrix Index	n	nnz	Left-Looking (hr)	Up-Looking	SLIP LU	Q-LDL
888	9801	87025	21.89	1	**N/A**	**N/A**
1330	7102	340200	20.14	0.99	**N/A**	**N/A**
760	4704	104756	18.50	1	**N/A**	**N/A**
887	9604	85264	17.68	0.99	**N/A**	**N/A**
50	4410	219024	14.69	1	**N/A**	**N/A**
46	3562	159910	12.92	1	1.23	**N/A**
791	8205	125567	12.28	0.99	**N/A**	**N/A**
35	2003	83883	8.78	1.01	**N/A**	**N/A**

				Relative Run Time		
Matrix Index	n	nnz	Left-Looking (hr)	Up-Looking	SLIP LU	Q-LDL
360	4515	97707	8.57	0.99	1.61	1.93
413	6867	98671	5.20	0.99	1.22	3.51
2211	2000	41906	4.83	0.98	0.44	**N/A**
759	2910	174296	3.92	0.99	3.93	5.85
1214	4098	72356	2.16	1	**N/A**	3.17
758	2146	72250	2.00	0.99	1.33	4.74
440	3363	99471	1.85	1.02	3.97	3.66
43	3600	26600	1.31	0.99	3.38	0.58
36	1806	63454	1.25	1	1.56	5.39

Table 2: Results for Instances Greater than 1 Hour

				Relative Run Time		
Matrix Index	n	nnz	Left-Looking (min)	Up-Looking	SLIP LU	Q-LDL
359	1440	44998	57.61	1	2.05	6.09
357	726	34518	37.09	1	0.75	7.58
757	1824	39208	36.34	1	4.74	6.60
411	2568	75628	33.60	1	3.89	2.85
1437	19998	99982	26.24	0.95	2.31	8.12
889	9801	87025	24.47	1	0.12	3.65
31	1083	18437	16.26	0.99	0.95	6.79
228	1919	32399	14.33	1	2.36	2.16
48	1922	30336	14.28	1	6.87	3.74
34	1473	34241	13.70	1	1.34	3.67
33	1473	34241	13.66	1	1.36	3.77
427	1821	52685	13.58	0.99	2.87	2.97
1331	7102	170134	13.19	1	3.40	1.28
408	2548	57308	11.39	1	3.21	1.16
1439	19994	79966	10.27	0.93	4.41	3.89
407	2410	54840	10.20	1.01	3.54	1.26
49	1224	56126	9.00	1	1.68	6.51
30	1074	12960	7.51	1	10.54	4.19
1911	1282	30644	6.84	1	1.14	1.61
339	588	21418	4.48	1	1	5.90
430	1733	22189	3.49	1	2.38	1.71
219	960	15844	3.22	0.99	1.44	2.10
358	1050	26198	2.30	1	5.36	3.08
223	729	4617	1.89	1.06	3.06	2.01
67	1473	19659	1.71	0.99	3.01	0.76
2210	700	12654	1.61	1	0.36	13.32
32	1086	22070	1.22	0.98	2.60	2.80
1328	500	28726	1.01	0.98	4.03	4.58

Table 3: Results for Instances Between 1 Min and 1 Hour

				Relative Run Time		
Matrix Index	n	nnz	Left-Looking (s)	Up-Looking	SLIP LU	Q-LDL
221	468	5172	58.150	1	1.99	0.31
28	420	7860	21.426	0.98	1.83	3.17
29	420	7860	21.025	0.99	1.86	3.23
2209	500	8478	19.073	1.01	0.56	14.42
62	420	7252	12.991	1	1.61	5.64
41	817	6853	12.838	0.98	2.47	3.31
222	675	3255	7.336	1	2.21	2.07
229	362	5786	5.094	1	1.33	3.62
929	415	2779	2.289	0.99	3.74	3.81
1401	2541	7361	2.173	0.92	7.65	0.01
4	685	3249	2.010	0.98	5.26	2.17
2208	300	4678	1.616	1	0.55	15.65
1	1138	4054	1.516	1.22	6.31	0.70
42	485	3135	1.219	1	2.12	3.61
75	15439	15439	1.014	1	1.36	0.02
875	306	2018	1.004	1	3.11	2.98
3	662	2474	0.991	0.96	6.40	2.20
876	306	2018	0.898	1.05	3.54	3.28
26	132	3648	0.649	0.81	1.73	5.12
1425	10001	49999	0.567	0.91	1.82	0.09
878	289	1377	0.538	1.12	1.97	4.38
206	147	2449	0.492	0.99	2.18	3.83
159	900	7744	0.450	0.98	0.40	5.86
27	153	2423	0.375	1.11	2.14	8.02
207	147	2441	0.348	1.03	2.32	5.13
2206	199	2873	0.241	1.12	0.93	17.23
2207	200	2890	0.240	1.16	0.83	16.42
2	494	1666	0.193	1.09	5.34	1.69
315	416	2312	0.104	1.13	3.02	0.09
1506	124	12068	0.101	0.94	1.46	11.35
2205	150	2040	0.097	1.02	1.20	14.99
73	3134	3134	0.094	0.76	0.88	0.06
24	66	4356	0.084	1.41	1.68	12.71
44	138	696	0.042	0.71	2.25	4.82
218	957	4137	0.028	1.22	2.64	0.26
76	1922	1922	0.028	1.04	1.08	0.10
220	100	594	0.021	0.63	2.56	5.74
23	48	400	0.015	1.70	2.02	7.97
63	1074	1074	0.014	1.04	0.74	0.15
66	1473	1473	0.012	1.64	1.26	0.22
873	48	306	0.009	0.96	0.87	5.97
877	289	1377	0.008	1.21	0.96	10.29
25	112	640	0.007	1.70	2.69	5.92
874	48	306	0.006	1.01	2.18	5.44
872	48	306	0.005	0.93	2.09	7.72
71	3600	3600	0.005	1.15	1.49	1.03
69	817	817	0.005	2.07	1.48	0.24
70	485	485	0.003	0.73	1.83	0.35

				Relative Run Time		
Matrix Index	n	nnz	Left-Looking (s)	Up-Looking	SLIP LU	Q-LDL
1939	3240	3240	0.002	0.83	1.39	2.74
61	420	420	0.002	1.28	0.72	0.30
436	27	279	0.001	1.17	1.98	8.15
217	237	1017	0.001	1.52	3.37	1.44
60	153	153	0.001	0.36	1.41	0.34
1438	18	82	0.001	1.04	1.03	4.66
57	66	66	0.000	0.50	1.37	0.30
1440	14	46	0.000	0.84	0.52	3.06
2204	20	158	0.000	1.45	2.05	3.01
2203	19	147	0.000	1.42	2.11	5.54
72	138	138	0.000	0.94	2.06	1.36

Table 4: Results for Instances Less than 1 Minute